

Riassunto Sistemi Operativi

lunedì 12 giugno 2017 15:59

Generalità di un SO

1.0 Cos'è un sistema operativo?

Un **sistema operativo** è un **programma**, o un **insieme di programmi**, che fornisce all'utente una **visione astratta e semplificata dell'hardware** e **gestisce in modo efficace ed efficiente le risorse** del sistema.
È l'**intermediario tra utente e hardware**.

1.1 Evoluzione dei SO

- **Sistemi "Batch"** (2^a gen.), In questo tipo di sistemi il compito unico del sistema operativo è quello di **trasferire il controllo da un job, appena terminato, al prossimo da eseguire del lotto, detto "batch"**.
La CPU è spesso **inattiva** a causa dell'attesa di eventi da parte dei job.
- **Sistemi multi-programmati** (3^a gen.), viene precaricato sul disco un insieme di job, detto **pool**. Il SO ha il compito di caricare un sottoinsieme di job appartenenti al **pool** e selezionare un job tra essi che verrà assegnato alla CPU. Qualora il job assegnato alla CPU si ponesse in attesa di un evento, il SO assegnerebbe la CPU ad un altro job. L'obiettivo della multiprogrammazione è la massimizzazione dell'utilizzo della CPU e della memoria.
- **Sistemi time-sharing** (4^a gen.), sistemi che implementano le seguenti caratteristiche:
 - **Multi-utenza**, il SO presenta ad ogni utente una macchina virtuale completamente dedicata
 - **Interattività**, il SO interrompe l'esecuzione di un job dopo un intervallo di tempo prefissato, assegnando la CPU ad un altro job.

1.2 Interruzioni

Le varie **componenti hardware e software interagiscono con il SO attraverso interruzioni asincrone** (*Interrupt*).

Ogni interruzione è **causata da un evento ed è associata ad un handler** per la gestione di tale evento.

Quando un **programma chiede l'esecuzione di un servizio del SO** parliamo di **System call**.

Alla ricezione di una interruzione il SO interrompe la sua esecuzione salvandone lo stato in memoria, attiva successivamente l'**handler** specifico dell'evento e infine ripristina lo stato salvato in memoria.

1.3 I/O

L'I/O in un sistema di elaborazione avviene tramite dei **controller, ovvero interfacce HW delle periferiche verso il bus di sistema**. Ogni controller è dotato di un **registro dati, dove memorizzare le informazioni da leggere e scrivere** e di alcuni **registri speciali, ove memorizzare le operazioni da eseguire (registro di controllo) e l'esito delle operazioni eseguite (registro di stato)**.

Il driver di dispositivo è un componente del SO che interagisce direttamente con il dispositivo e la sua struttura è strettamente dipendente da esso.

1.4 Protezione

Nei SO che prevedono multiprogrammazione e multi-utenza sono necessari alcuni meccanismi di protezione.

È **necessario impedire al programma in esecuzione di accedere ad aree di memoria esterne al proprio spazio**.

Per garantire protezione, molte CPU prevedono un duplice modo di **funzionamento (dual-mode)**:

- **Kernel mode**, può eseguire istruzioni privilegiate. Il **SO esegue in modo kernel** a differenza dei **processi utente che per eseguire istruzioni privilegiate si avvalgono di System call**.
- **User mode**, programmi utente

1.5 Struttura

Le componenti di un sistema operativo possono essere organizzate in modi differenti all'interno del SO:

- **Struttura Monolitica**, il SO è costituito da un unico modulo che realizza le varie componenti. Questo assicura un basso costo di interazione tra le varie componenti, ma il SO è estremamente complesso e poco adattabile.
- **Struttura Modulare (stratificato)**, Le varie componenti del SO vengono organizzate in moduli caratterizzate da interfacce ben definite.
Ogni strato ha un insieme di funzionalità che vengono mostrate allo strato superiore mediante interfacce. Questa struttura permette l'**astrazione** e la **modularità** ma implementa una **organizzazione gerarchica** delle componenti (che non sempre è possibile) e una **scarsa efficienza data dal costo di attraversamento dei vari livelli**.
- **Struttura a Microkernel**, la **struttura del nucleo (kernel)** è ridotta al minimo (solo funzionalità base). Il resto del sistema operativo è rappresentato da processi utente.
È un sistema **molto affidabile e personalizzabile** ma **non molto efficiente a causa delle molte System call**.

1.6 Macchina Virtuale

Le **MV sono l'evoluzione dell'approccio a livelli**. Virtualizzano sia **HW che il kernel del SO**.

Su una stessa macchina fisica danno l'illusione di elaboratori multipli, ciascuno in esecuzione sul suo processo privato e con la sue risorse private, messa a disposizione dal **kernel del SO**, che può essere diverso per processi diversi.

Una singola piattaforma HW può essere condivisa da più SO, ognuno dei quali è installato su una diversa macchina virtuale. Il disaccoppiamento è realizzato da un componente detto **Virtual Machine Monitor di sistema** che ha il **compito di consentire la condivisione da parte di più MV di un'unica piattaforma HW**.

In particolare, una **VMM di sistema** integra le sue funzionalità in un **SO leggero**, costituendo un sistema posto **direttamente sopra l'HW**.

Una **VMM ospitata** è invece una **MV che viene installata come applicazione sopra un SO esistente**, opera nello spazio utente e accede all'HW tramite **System call**.

Processi e Thread

2.0 Processo

Un **processo** è l'unità di esecuzione all'interno del SO.

Solitamente l'esecuzione è sequenziale, ma un **SO multi-programmato** consente l'esecuzione concorrente di più processi.

Un processo durante la sua esistenza può trovarsi in vari stadi:

- **Init**, stato di transizione durante il quale il processo **viene caricato in memoria e vengono inizializzati i suoi dati**
- **Ready**, processo **pronto ad acquisire la CPU**
- **Running**, il processo sta **utilizzando la CPU**
- **Waiting**, processo è in **sospeso in attesa di un evento**
- **Terminated**, stato di transizione **tra la terminazione e la deallocazione del processo**

Il SO associa ad ogni processo una struttura dati detta **PCB (Process Control Block)**.

Il PCB contiene tutte le informazioni relative al processo.

2.1 Scheduling

Lo **scheduling** è l'attività mediante la quale il SO effettua delle scelte tra i processi, riguardo al loro caricamento in memoria centrale o l'assegnazione alla CPU.

In generale il SO effettua 3 diverse attività di scheduling:

- **Scheduling a breve termine (o di CPU)**, si occupa di **selezionare il processo da assegnare alla CPU e di effettuare il context switch** (cambio di contesto), ovvero la fase in cui l'uso della CPU viene commutato da un processo ad un altro.
- **Scheduling a medio termine (o swapping)**, gestisce il **trasferimento temporaneo in memoria secondaria di processi (o parti di essi)**, in modo da consentire il caricamento di altri processi in memoria centrale
- **Scheduling a lungo termine**, seleziona i programmi da eseguire dalla memoria secondaria per caricarli in memoria centrale. È una componente importante dei sistemi batch e dei sistemi multi-programmati, ma **non è presente nei sistemi time sharing** in quanto è l'utente a stabilire i processi da eseguire.

2.2 Operazioni sui processi

Ogni SO prevede dei **meccanismi per la gestione dei processi**:

- **Creazione**
- **Interazione tra processi**
- **Terminazione**

Queste sono operazioni privilegiate e vengono **eseguite in modo kernel**.

2.3 Thread (o processi leggeri)

Un **thread (o processo leggero)** è una unità di esecuzione che **condivide codice e dati con gli altri thread ad esso associati**.

L'**insieme di thread che riferiscono allo stesso codice e dati è detto Task**.

Un processo pesante equivale ad un task con un singolo thread.

I **thread hanno come vantaggio la condivisione di memoria** con altri thread dello stesso Task, un **costo minore di contest switch** in quanto il PCB dei thread non contiene nessuna informazione relativa a codice e dati.

I **thread garantiscono una minore protezione** in quanto thread di uno stesso task possono modificarsi i dati a vicenda.

2.4 Interazione tra processi

2 processi si possono dire:

- **Processi interagenti**: se l'esecuzione di uno **è influenzata** dall'esecuzione dell'altro e/o viceversa. In particolare l'interazione può essere:
 - **Cooperazione**, interazione prevedibile e desiderata per il raggiungimento di un fine comune
 - **Competizione**, interazione prevedibile ma indesiderata
 - **Interferenza**, interazione non prevista e non desiderata
- **Processi indipendenti**, se l'esecuzione di uno **non è influenzata** dall'esecuzione dell'altro e/o viceversa.

L'interazione tra processi può avvenire mediante:

- **Memoria condivisa** (Ambiente globale), Il SO consente ai **processi di condividere variabili** e l'interazione avviene tramite esse.
- **Scambio di messaggi** (Ambiente locale), i processi non condividono variabili e **interagiscono mediante la trasmissione e la ricezione di messaggi**.
Può essere **diretta, specificando il processo che deve ricevere il messaggio** (e conoscendo il processo che spedisce il messaggio nel caso del ricevente), o **indiretta con modello a mailbox**.

2.5 Processi in UNIX

In UNIX ogni processo è il proprio spazio di indirizzamento **completamente locale e non condiviso** (ambiente locale).

In UNIX ogni processo può entrare in **2 ulteriori stati**, oltre quelli generali, che sono:

- **Zombie**, il processo è **terminato ma in attesa che il padre ne rilevi lo stato di terminazione**
- **Swapped**, il processo o parte di esso è temporaneamente trasferito in memoria secondaria dallo scheduler a medio termine.

Il **codice (text)** di dei processi è **rientrante**, quindi più processi possono condividere lo stesso codice.

Codice e dati sono separati (modello a codice puro).

Il SO gestisce una struttura dati globale in cui sono contenuti i **puntatori ai codice (text)** utilizzati, ed eventualmente condivisi da altri processi. Questa struttura è detta **text table**.

L'elemento della text table è detto **text structure** e **contiene un puntatore al codice** e il numero di processi che lo condividono.

Scheduling della CPU

3.0 Generalità

Lo **scheduler della CPU** è quel componente del SO che si occupa di **selezionare dalla coda dei processi Ready il prossimo processo al quale assegnare l'uso della CPU**.

La gestione della coda dei processi *Ready* è realizzata mediante *algoritmi di scheduling*. In particolare abbiamo 2 categorie di algoritmi di scheduling:

- **Senza prelazione (o non pre-emptive)**, la CPU rimane allocata al processo *Running* finché esso non si sospende volontariamente o termina.
- **Con prelazione (o pre-emptive)**, il processo *Running* può essere prelazonato, cioè sospeso dal SO per assegnare la CPU ad un nuovo processo. I sistemi a divisione di tempo implementano sempre queste politiche di scheduling.

3.1 Criteri di scheduling

Per confrontare i vari algoritmi di scheduling vengono considerati alcuni indicatori di performance.

In generale deve essere massimizzato l'utilizzo della CPU e il numero di processi completati nell'unità di tempo e minimizzati il tempo di attesa di un processo nella *Ready queue*, il tempo tra l'inizio di un job e il suo completamento e l'intervallo tra la sottomissione di un job e la sua prima risposta.

3.2 Algoritmi di scheduling

- **FCFS (first come first served)**, la coda dei processi è gestita in modalità FIFO. È un algoritmo senza prelazione ed è impossibile influire sull'ordine dei processi
- **SJF (shortest job first)**, per ogni processo *Ready* viene stimato il tempo di esecuzione e viene schedulato per primo il processo con il *CPU burst* stimato più breve. Può essere sia con, che senza, prelazione
- **Round Robin**, tipicamente utilizzato nei sistemi *time sharing*, gestisce la coda come una *FIFO circolare* e ad ogni processo viene allocata la CPU per un intervallo di tempo costante detto *time slice*, dopo questo intervallo il processo viene reinserito in coda e la CPU passa al processo successivo.
- **Con priorità**, ad ogni processo viene assegnata una priorità (definita dal SO o esternamente).
La priorità di un processo può essere **costante o variata dinamicamente**.
Questo algoritmo presenta il **problema della starvation dei processi con priorità bassa**.
La soluzione è la **modifica dinamica delle priorità**, come ad esempio di decrescere della priorità al crescere del tempo assegnato alla CPU.

Spesso nei SO reali vengono combinati diversi algoritmi di scheduling.

File System

4.0 Generalità

Il **file system** è quella componente del SO che fornisce i meccanismi di accesso e memorizzazione delle informazioni allocate in memoria secondaria.

Realizza i concetti astratti di:

- File, unità logica di memorizzazione
- Direttorio, insieme di file
- Partizione, insieme di file associato ad un particolare dispositivo fisico

L'organizzazione del file system è divisa su livelli, in particolare:

- **Struttura logica**, presenta alle applicazioni una visione astratta delle informazioni memorizzate
- **Accesso**, definisce e realizza i meccanismi di accesso al contenuto del file, in particolare
 - Definisce il **concetto di unità di trasferimento** da/verso file: **record logico**
 - Definisce i **metodi di accesso** (*sequenziale, diretto, ad indice*)
 - Realizza i **meccanismi di protezione**
- **Organizzazione fisica**, rappresentazione di file e directory sul disco e **mapping dei record logici sui blocchi**
- **Dispositivo virtuale, vista astratta del dispositivo** che appare come una sequenza di **blocchi di dimensione costante**

4.1 File

Il file è un insieme di informazioni. Ogni file è individuato da (almeno) un nome simbolico ed è caratterizzato da un insieme di attributi:

- Tipo
- Indirizzo
- Dimensione
- Data e ora di creazione e/o modifica
- Utente proprietario
- Protezione (diritti di accesso)

Tutti gli attributi sopra elencati sono contenuti all'interno di una struttura dati chiamata **descrittore del file**.

Ogni descrittore deve essere memorizzato in modo persistente.

4.1.1 Operazioni sui file

Il **compito del SO** è quello di **consentire l'accesso on-line ai file**, ovvero di rendere immediatamente visibile le modifiche effettuate da un processo su un file agli altri processi.

Le operazioni più comuni sono:

- **Creazione**, allocazione di un file in memoria secondaria e inizializzazione dei suoi attributi
- **Lettura** di record logici dal file
- **Scrittura**, inserimento di nuovi record logici all'interno del file
- **Cancellazione**, eliminazione del file dal file system

Per garantire una maggiore efficienza, il **SO registra i file attualmente in memoria** in una struttura detta **tabella dei file aperti** ed effettua lo spostamento dei file aperti in memoria centrale (*memory mapping*).

Le operazioni necessarie sono:

- **Apertura**, introduzione di un nuovo elemento nella tabella dei file aperti ed eventuale *memory mapping*.
- **Chiusura**, salvataggio del file in memoria secondaria e rimozione dell'elemento corrispondente dalla tabella dei file aperti.

4.1.2 Struttura interna dei file

Ogni **dispositivo di memorizzazione secondaria** viene diviso in **blocchi di dimensione fissa** (*record fisici*), essi sono le unità di trasferimento fisico nelle operazioni di I/O.

L'utente vede il file come un insieme di **record logici**, ovvero le unità di trasferimento logico nelle operazioni di accesso ai file. I **record logici** hanno sempre **dimensione variabile**.

È compito del SO stabilire una corrispondenza tra blocchi fisici e record logici.

Di solito la **dimensione dei blocchi** è superiore a quella dei **record logici**, avviene quindi l'**impacchettamento di più record logici all'interno dei blocchi**.

4.1.3 Metodi di accesso ai file

L'accesso ai file può avvenire secondo 3 modalità:

- **Accesso sequenziale**, il file è una **sequenza di record logici**. Per accedere ad un particolare record è necessario accedere prima ai record che lo precedono.
- **Accesso diretto**, il file è un **insieme di record logici numerati**. Molto **utile quando si ha a che fare con file molto grandi per leggere/modificare poche informazioni**.
- **Accesso ad indice**, ad ogni file viene **associata una struttura dati contenente l'indice delle informazioni contenute**. Per accedere ad un record logico si esegue una **ricerca nell'indice**.

4.2 Directory

La **directory** (o **direttorio**) è uno strumento per **organizzare i file** all'interno del SO.

La **struttura logica** delle directory può variare a seconda del SO. In particolare le realizzazioni logiche principali sono:

- **Struttura ad 1 livello**, un **solo directory per ogni file system** (difficile separare file per una multi-utenza).
- **Struttura a 2 livelli**, il **primo livello contiene una directory per ogni utente del sistema**, mentre il **secondo livello (directory utenti)** contengono i file dei rispettivi utenti.
- **Struttura ad albero**, organizzazione gerarchica a [Equazione] livelli. Ogni directory può contenere sia file che directory. Al primo livello si trova il **direttorio radice**.
- **Struttura a grafo aciclico**, estende la **struttura ad albero** con la possibilità di inserire **link differenti allo stesso file**.

4.3 Organizzazione fisica del file system

Il SO si occupa anche della **realizzazione del file system sui dispositivi di memorizzazione secondaria**.

In particolare le principali tecniche di allocazione dei **blocchi** (insieme di **record logici** contigui) sono:

- **Allocazione contigua**, ogni file è mappato su un insieme di blocchi fisicamente contigui dando la possibilità di accesso sequenziale e diretto.
 - **Vantaggi**: basso costo di ricerca di un blocco e possibilità di accesso sequenziale e diretto
 - **Svantaggi**: difficile individuare lo spazio libero per l'allocazione di nuovi file.
Frammentazione esterna, man mano che si riempie il disco, rimangono zone contigue sempre più piccole e a volte inutilizzate.
- **Allocazione a lista**, i blocchi sui quali viene mappato ogni file sono organizzati a **lista concatenata**. I puntatori ai blocchi sono distribuiti sul disco.
 - **Vantaggi**: Non c'è *frammentazione esterna* e basso costo di allocazione.
 - **Svantaggi**: necessità di dedicare spazio ai puntatori (maggiore spazio occupato), difficile rendere possibile l'accesso diretto e costo elevato per la ricerca di un blocco.
- **Allocazione ad indice**, ad ogni file è associato un blocco (**blocco indice**) in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file.
 - **Vantaggi**: Non c'è *frammentazione esterna*, basso costo di allocazione e possibilità di accesso diretto.
 - **Svantaggi**: possibile scarso utilizzo dei **blocchi indice**.

Esistono SO che adottano più di un metodo di allocazione.

4.4 File system in UNIX

Il file system di UNIX è composto da 3 tipologie distinte di file:

- **File ordinari**
- **Directory**
- **Dispositivi fisici** (file speciali contenuti nel direttorio /dev)

I file vengono allocati con la tecnica di **allocazione ad indice** a più livelli di indirizzamento.

Il file system è partizionato in 4 regioni:

- **Boot block**, contiene le procedure di inizializzazione del SO.
- **Super block**, fornisce i limiti delle 4 regioni, il puntatore alla lista dei blocchi liberi etc...
- **Data blocks**, è l'area in cui memorizzare il file. Contiene i blocchi allocati e quelli liberi.
- **i-List**, contiene la lista di tutti i descrittori dei file, detti **i-node**, direttori e dispositivi presenti nel file system, accessibili attraverso l'indice **i-number**.

L'**allocazione del file non è su blocchi fisicamente contigui**; il suo **i-node** contiene i puntatori ai blocchi.

Ogni file è accessibile secondo 3 diverse modalità: **scrittura**, **lettura** ed **esecuzione**.

Il **proprietario di un file può concedere o negare il permesso di accedere al file ad altri utenti**.

Esiste un **utente privilegiato**, detto utente **root**, che ha **accesso ad ogni file del sistema**.

Ad ogni file (nel suo **i-node**) sono associati **12 bit di protezione**. I primi 9 bit sono i permessi per l'utente proprietario, utenti nel gruppo e i restanti utenti; gli ultimi 3 bit di permessi sono per i file eseguibili.

Al processo che esegue un file eseguibile è associato dinamicamente un User ID e un Group ID.

Anche i **direttori sono rappresentati da file nel file system in UNIX**.

Ogni file-direttorio contiene un insieme di record logici costituiti dal **nome relativo** di ogni file o sotto-direttorio contenuto e il relativo **i-number** che lo identifica.

4.4.1 Accesso ai file in UNIX

UNIX implementa **accesso sequenziale** ai file. Un **I/O pointer** (puntatore al file) registra la posizione corrente nel file.

Ad ogni processo è associata una **tabella dei file aperti di processo** di dimensione limitata ([Equazione]).

Ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero detto **file descriptor** (**fd**). I **fd** [Equazione] , rappresentano rispettivamente **stdin** (0), **stdout** (1), **stderr** (2).

La tabella dei file aperti del processo è allocata nella sua **user structure**.

Per realizzare l'accesso ai file, il SO utilizza 2 strutture dati globali, allocate nell'area dati del kernel:

- **Tabella dei file attivi**, per ogni file aperto, contiene una copia del suo **i-node**.

- **Tabella dei file aperti di sistema**, ha un elemento per ogni operazione di apertura (anche dello stesso file) relativi ai file aperti (e non ancora chiusi). Ogni elemento contiene *I/O pointer* (indica la posizione corrente all'interno del file) e un puntatore all'*i-node* del file nella tabella dei file attivi.

Gestione della memoria centrale

5.0 Generalità

A livello HW ogni sistema ha disposizione un **unico spazio di memoria** accessibile direttamente da CPU e altri dispositivi.

I compiti del SO nella gestione della memoria sono:

- **Accesso alla memoria centrale**, deve svolgere *load* e *store* di dati e istruzioni. Gli indirizzi possono essere *simbolici*, *logici* e *fisici*. Ogni processo dispone di un proprio spazio di indirizzamento logico che viene allocato nella memoria fisica.
- **Binding degli indirizzi**, ossia l'associazione ad ogni indirizzo *logico* o *simbolico* di un indirizzo *fisico*. Può essere svolto staticamente (a tempo di compilazione) o a tempo di caricamento.
- **Caricamento/collegamento dinamico**, il cui obiettivo è l'ottimizzazione della memoria. Funzioni possono essere usate da più processi simultaneamente e il SO deve gestire gli accessi allo spazio di altri processi.
- **Gestire spazi di indirizzi logici** di dimensione maggiore allo spazio fisico. L'**overlay** è la soluzione al problema in cui la memoria disponibile non sia sufficiente ad accogliere codice e dati di un processo, esso **mantiene in memoria codice e dati usati più di frequente e che sono necessari nella fase corrente**.
Codice e dati vengono quindi divisi in overlay che vengono caricati e scaricati dinamicamente.

5.1 Tecniche di allocazione in memoria centrale

Codice e dati dei processi possono essere allocati in memoria centrale in 2 modi:

- **Allocazione contigua**
- **Allocazione non contigua**

5.1.1 Allocazione contigua

Nel caso dell'allocazione contigua, abbiamo a sua volta 2 implementazioni principali:


- Allocazione **contigua a partizione singola**, la parte di memoria per l'allocazione dei processi non è partizionata e **solo un processo alla volta** può essere allocato in memoria (**no multi-programmazione**).
- Allocazione **contigua a partizioni multiple**, c'è multiprogrammazione e quindi necessità di proteggere codice e dati di ogni processo. Ad ogni processo viene associata una partizione.

Le partizioni possono essere:

- **Fisse**, la **dimensione di ogni partizione è fissata a priori** e per ogni processo viene cercata una partizione libera di sufficiente dimensione.
Ha come problema il crearsi di **frammentazione interna dovuta al sottoutilizzo della partizione**, e un grado di multiprogrammazione limitato al numero di partizioni.
- **Variabili**, le partizioni vengono **allocate dinamicamente e dimensionate in base alla dimensione del processo**. Quando un processo viene schedato dal SO, esso cercherà un'area di memoria abbastanza grande dove allocare dinamicamente la partizione associata.
Viene **eliminata la frammentazione interna** in quanto le partizioni sono della dimensione richiesta ma questa tecnica ha delle **problematiche** che risiedono nella **scelta dell'area migliore** in cui allocare il processo e nella **frammentazione esterna**.

5.1.2 Allocazione non contigua

Per eliminare la **frammentazione esterna** si passa all'**allocazione non contigua**, che si divide in:

-  **Paginazione**, lo spazio fisico viene considerato come un insieme di pagine, dette *frame*, di dimensione costante, sulle quali mappare porzioni dei processi da allocare.

In questa modalità non si parla più di **frammentazione esterna** ed è **possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo**.

Il binding tra indirizzi logici e indirizzi fisici può essere realizzato mediante una **tabella delle pagine**, che ad ogni pagina logica associa la pagina fisica corrispondente.

Dato che **la tabella può essere molto grande**, e la traduzione della corrispondenza tra i due indirizzi dev'essere molto veloce, ci sono varie possibili soluzioni:

- Su registri CPU
- In memoria centrale
- Memoria centrale + cache

Quando lo spazio logico di indirizzamento di un processo è molto esteso si ha un elevato numero di pagine e una tabella delle pagine di grandi dimensioni. In questo caso si può utilizzare la **paginazione su più livelli**, che consiste nell'**allocazione non contigua con paginazione anche della tabella delle pagine**.

Il tempo di accesso aumenta al crescere dei livelli di paginazione.

Per limitare l'uso della memoria alcuni SO utilizzano un'unica struttura dati globale detta **tabella delle pagine invertita**, che ha un elemento per ogni frame. Ogni **"pagina invertita"** rappresenta un frame e contiene l'identificativo (*pid*) del processo a cui è assegnato il frame. Il tempo di ricerca in questa soluzione è alto e la struttura rende **difficile la rientranza, ovvero la condivisione di un frame tra processi diversi**.

- **Segmentazione**, si basa sul partizionamento dello spazio logico degli indirizzi di un processo in segmenti caratterizzati da nome e lunghezza.
Ogni **segmento viene allocato in memoria in modo contiguo** e ad ognuno di essi il SO associa un intero attraverso il quale lo si può riferire.
È presente una **tabella dei segmenti** che ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica.
La segmentazione (più segmenti per processo) è l'evoluzione della tecnica di **allocazione contigua a partizioni variabili** (1 segmento per processo).
Il problema principale è la **frammentazione esterna**, che viene risolta con l'adozione di politiche adatte alla situazione.

Paginazione e Segmentazione possono essere unite nella **segmentazione paginata**, in cui lo spazio logico è segmentato e ogni segmento è suddiviso in pagine. Con questo metodo si **elimina la frammentazione esterna** e **non è necessario mantenere in memoria un intero segmento** ma basta caricare solo le pagine necessarie. Sono presenti una **tabella dei segmenti** e una **tabella delle pagine per ogni segmento**.

5.2 Memoria Virtuale

La dimensione della memoria può rappresentare un vincolo importante riguardo alla dimensione dei processi e al grado di multiprogrammazione. La memoria virtuale è una tecnica che permette l'esecuzione di processi non completamente in memoria.

Questa tecnica ha molti vantaggi:

- La dimensione dello spazio logico degli indirizzi è indipendente dall'estensione della memoria.
- Il grado di multiprogrammazione è indipendente dalla dimensione della memoria fisica.
- Il caricamento di un processo e il suo *swapping* hanno un costo ridotto.
- Il programmatore non deve preoccuparsi dei vicoli relativi alla dimensione della memoria.

5.2.1 Paginazione su richiesta

Solitamente la memoria virtuale è realizzata mediante tecniche di **paginazione su richiesta** in cui tutte le pagine di ogni processo risiedono in memoria secondaria e durante l'esecuzione alcune di esse vengono trasferite all'occorrenza in memoria primaria. È presente un *pager*, ovvero una componente del SO che si occupa del trasferimento delle pagine. È detto **pager lazy** (*pager prigo*) un pager che **trasferisce in memoria centrale una sola pagina se ritenuta necessaria**. Se si utilizza la *paginazione su richiesta*, l'esecuzione di un processo può richiedere *swap-in* del processo.

Una pagina dello spazio logico di un processo può essere allocata in memoria primaria o secondaria e lo si distingue dai bit di validità presenti nella tabella delle pagine. La pagina può essere anche invalida, nel caso non esista lo spazio logico del processo. Nel caso di pagina invalida viene mandata una **interruzione di page fault**.

In seguito al *page fault*, se è necessario caricare una pagina in memoria primaria, potrebbero non esserci *frame* liberi. In questo caso si utilizza la **sorrallocazione** ovvero la **sostituzione in memoria primaria di una "pagina vittima"** con la pagina richiesta.

La *"pagina vittima"* viene quindi copiata in memoria secondaria, nel caso sia stata modificata.

Si introduce quindi la necessità di introdurre efficaci ed efficienti **algoritmi di sostituzione** che siano in grado di **sostituire quelle pagine la cui probabilità di accesso a breve termine è bassa**. I principali sono:

- **LFU** (*least frequently used*), viene sostituita la **pagina che è stata usata meno spesso**.
- **FIFO** (*first in, first out*), viene sostituita la **pagina che è da più tempo in memoria**.
- **LRU** (*least recently used*), viene sostituita la **pagina che è stata usata meno recentemente**.

5.2.1 Pre-paginazione e working set

È possibile che il processo impieghi più tempo per la sua paginazione che per la sua esecuzione; in questo caso si parla di fenomeno di *thrashing*. Per contrastarlo si usano tecniche di gestione della memoria basate sulla **pre-paginazione**, ovvero nel prevedere il set di pagine, detto **working set**, il cui processo da caricare avrà bisogno nella fase successiva di esecuzione.

Un processo, in una sua fase di esecuzione, usa un sottoinsieme relativamente piccolo delle sue pagine logiche, che varia nel tempo.

Si dice **località spaziale** l'alta probabilità di accedere a locazioni vicine nello spazio a locazioni appena accedute, e **località temporale** l'alta probabilità di accedere a locazioni accedute di recente.

Nella tecnica della **pre-paginazione con working set** quando si carica un processo, si carica il suo **working set iniziale**, che verrà **aggiornato dinamicamente** in base al principio di **località temporale**.

5.3 Gestione della memoria in UNIX

UNIX utilizza la *segmentazione-paginata* e la *memoria virtuale* tramite *paginazione su richiesta senza working set*.

Inoltre l'allocazione di ogni segmento non è contigua.

La *sostituzione delle pagine*, tramite lo *swapper di pagine*, viene attivata dallo *scheduler* quando il numero totale di *frame* liberi è considerato insufficiente.

Ogni *frame* ha dimensione costante.

Programmazione concorrente ad ambiente globale

6.0 Generalità

In una macchina ad ambiente globale le comunicazioni attraverso risorse condivise devono essere sincronizzate negli accessi. Esistono diverse tecniche di sincronizzazione gestite direttamente dal *kernel* (nucleo) del SO.

Ogni applicazione può essere rappresentata come un insieme di 2 componenti:

- **Processi** (componenti attivi)
- **Risorse** (componenti passivi), sono oggetti fisici o logici che vengono utilizzati da processo per il suo completamento.

Le risorse sono raggruppate in **classi di risorse**. Ogni classe identifica **l'insieme delle operazioni che un processo può svolgere sulle tutte risorse di quella classe**.

Le risorse possono essere **private** o **globali** a seconda che possano essere accedute e modificate da solo un processo o meno.

6.1 Deadlock

Un insieme di processi è in **deadlock**, o **blocco critico**, se **ogni processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo dell'insieme** (stallo).

Queste 4 sono le condizioni necessarie per il *deadlock*:

- **Mutua esclusione**
- **Possesso e attesa**
- **Impossibilità di prelazione**
- **Attesa circolare**

Se anche una delle quattro condizioni viene a mancare non c'è *deadlock*.

Le situazioni di *deadlock* devono essere evitate o prevenendo il *blocco critico* o curandolo:

- **Prevenzione**, che può essere *statica*, nel caso vengano posti vincoli ad almeno una delle 4 condizioni citate, o *dinamica*, se il processo prima di cedere una risorsa controlla il possibile verificarsi del *deadlock*.
- **Rilevazione/Ripristino**, il SO rileva il *deadlock* e attua un algoritmo di ripristino.

6.2 Mutua esclusione

Il **problema della mutua esclusione nasce quando più di un processo alla volta può avere accesso a risorse comuni**.

La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle risorse comuni non si sovrappongano nel tempo.

In particolare, una istruzione che opera su un dato è detta **indivisibile** (o *atomica*) se durante la sua esecuzione da parte di un processo, il dato non è accessibile da altri processi.

La sequenza di istruzioni con le quali un processo accede e modifica risorse comuni si dice **sezione critica**.

La **regola di mutua esclusione** impone che una sola sezione critica di una classe può essere in esecuzione in ogni istante.

Lo schema generale di risoluzione è assicurare un **prologo** e un **epilogo** all'inizio e alla fine di ogni sezione critica.

Il **prologo** in particolare è quell'insieme di istruzioni con il quale un **processo richiede l'autorizzazione all'uso esclusivo di una risorsa**.

L'**epilogo** è l'insieme di istruzioni che vengono eseguite al completamento della **sezione critica** dove il **processo dichiara libera la sezione critica**.

Tra le tante soluzioni adottate l'**algoritmo di Dekker** è **starvation-free**.

Con il termine **starvation** si intende un fenomeno quando un processo attende un tempo infinito per l'acquisizione di una risorsa. Mentre per **starvation-free** si intende un algoritmo che garantisce ad ogni processo di accedere alla **sezione critica per un tempo finito**.

Thread in Java

7.0 Generalità

Un **thread** condivide codice, dati e spazio di indirizzamento con gli altri **thread** associati, ha però stack e program counter privati. Ad ogni programma Java corrisponde l'esecuzione di un **task** (processo), contenente almeno un singolo **thread**, corrispondente al metodo **main()** sulla Java Virtual Machine.

È comunque possibile creare dinamicamente **thread** che eseguono concorrentemente all'interno del programma.

Essi sono implementabili o estendendo la classe **Thread** o implementando l'interfaccia **Runnable**.

Il ciclo di vita di del thread può essere schematizzato in 4 fasi:

- **New thread** subito dopo la creazione dell'oggetto.
- **Runnable** eseguibile ma potrebbe non essere in esecuzione.
- **Not runnable** stato nel quale finisce quando non può essere messo in esecuzione a causa di metodi quali **wait()**, **suspend()**, **sleep()** o perché in attesa di una operazione di **I/O**.
- **Dead** stato finale in cui giunge per "morte naturale" o perché è stato invocato il metodo **stop()**.

Le interazioni tra i **thread** avvengono mediante oggetti in comune, distinguiamo le interazioni:

- **Competitive**: Mutua esclusione
- **Cooperative**: Semafori, Variabili condizione

7.1 Semafori

Il **semaforo** è uno strumento di sincronizzazione che permetti di risolvere qualunque problema di sincronizzazione tra **thread** nel modello ad ambiente globale. Viene **implementato solitamente dal kernel del SO**.

Un **semaforo** è un dato astratto rappresentato da un intero non negativo a cui è possibile accedere solo tramite le operazioni **p()** e **v()**.

p (s):	while(!s) s--
v (s):	s++

Le due operazioni sono **atomiche**. Il valore del semaforo viene modificato da un processo alla volta.

Utilizzando i semafori, la decisione se un processo può proseguire l'esecuzione dipende solo dal semaforo e la scelta del processo da risvegliare avviene tramite algoritmo di FIFO.

In problemi di sincronizzazione più complessi, invece, la decisione se un processo possa proseguire l'esecuzione dipende dal verificarsi di una **condizione di sincronizzazione**.

Per ovviare a questi problemi, si utilizzano costrutti linguistici più di alto livello come i **monitor**.

7.2 Monitor

Il **monitor** è un costrutto sintattico che **associa un insieme di operazioni dette entry ad una struttura dati comune** a più **thread**, tale che **le operazioni entry siano le sole operazioni permesse su quella struttura e siano mutualmente esclusive**.

Le variabili locali sono accessibili solo all'interno del **monitor**, così come le operazioni dichiarate non **public**.

Ad ogni monitor è associata una risorsa, lo scopo è di controllare gli accessi in accordo a determinate politiche.

L'accesso avviene mediante 2 livelli di sincronizzazione:

- Un **thread** alla volta può accedere alle risorse del **monitor** tramite le operazioni **entry**
- Controllo dell'ordine con i quali i **thread** hanno accesso alle risorse del monitor in base ad una **condizione di sincronizzazione** che ne assicura l'ordinamento.

Gestione I/O

8.0 Compiti del gestore di I/O

1. **Nascondere** al programmatore i dettagli delle interfacce hardware dei dispositivi.
2. **Omogeneizzare la gestione di dispositivi diversi**.
3. **Gestire i malfunzionamenti** che si possono verificare durante un trasferimento di dati.

4. **Definire lo spazio dei nomi** (*naming*) con cui vengono identificati i dispositivi, uniformandolo anche con il *naming* del file system.
5. **Garantire la corretta sincronizzazione** tra ogni processo applicativo che ha attivato un trasferimento dati e l'attività del dispositivo.
 - **Gestione sincrona**, il processo si blocca fino al termine del trasferimento
 - **Gestione asincrona**, il processo attiva il trasferimento e procede nella sua esecuzione

8.1 Architettura del sottosistema di I/O

La struttura del sottosistema di I/O è generalmente stratificata. I 2 livelli più importanti sono:

- **Livello indipendente dai dispositivi**, implementa le funzioni di:
 - **Naming**
 - **Buffering**, per ogni operazione di I/O il SO riserva un'area di memoria detta *buffer* per contenere i dati oggetto del trasferimento a causa della differenza di velocità tra processo e periferica che provocherebbe un disaccoppiamento e della quantità dei dati da trasferire (che nei dispositivi a blocchi potrebbe essere inferiore alla dimensione del blocco).
 - **Gestione dei malfunzionamenti**, prevede 2 approcci agli eventi anomali:
 - Risoluzione del problema
 - Gestione parziale e propagazione al livello superiore (applicativo)

I tipi di eventi che si possono generare sono:

 - Eventi propagati da un livello inferiore
 - Eventi generati a questo livello
- **Allocazione dei dispositivi ai processi applicativi**, deve gestire politiche allocazione (nel caso di dispositivo condiviso da più processi: mutua esclusione)
- **Livello dipendente dai dispositivi** (*device drivers*), deve fornire i gestori dei dispositivi (*device drivers*) e offrire al livello superiore delle funzioni di accesso ai dispositivi. Contiene tutto il codice dipendente dal dispositivo. In particolare la **gestione dei dispositivi** può implementata nei seguenti modi:
 - **A controllo di programma**, metodo non adatto per i sistemi multi-programmati, a causa dei cicli di attesa attiva, in quanto viene interrogato in maniera ciclica e continua del dispositivo
 - **A interruzione**, per evitare l'attesa attiva ad ogni dispositivo viene assegnato un semaforo ed è il controller del dispositivo a dare il segnale di interruzione al SO significante che è pronto per l'I/O
 - **In DMA**, è una alternativa che evita il sovraccarico della CPU rendendo possibile l'accesso "diretto" alla memoria tramite un controllore DMA. In questo modo viene bypassata la CPU e i suoi *buffer*

8.2 Gestione e organizzazione fisica dei dischi

Un blocco fisico su un disco è descritto da 3 parametri:

- **F**, numero della faccia
- **T**, numero della traccia in ambito della faccia
- **S**, numero settore all'interno della faccia

Tutti i settori che compongono un disco (o un pacco di dischi), possono essere trattati come un **array lineare di blocchi**, indicando con [Equazione] il numero di tracce per faccia e con [Equazione] il numero di settori per traccia.

Per **ridurre il tempo medio di trasferimento di un settore**, si può intervenire in 2 modi:

- **Criteri con cui i dati sono memorizzati sul disco** (metodi di allocazione dei file)
- **Criteri con cui servire le richieste di accesso** (politiche di *scheduling* delle richieste)

Nella valutazione del tempo medio di attesa di un processo è necessario tenere conto anche il tempo durante quale il processo attende che la sua richiesta venga servita.

Le richieste possono essere gestite secondo diverse politiche:

- **FCFS**, richieste servite rispettando il tempo di arrivo.
- **SSTF**, seleziona la richiesta con il tempo di seek minimo a partire dalla posizione attuale della testina del disco. Può portare a situazioni di *starvation* delle richieste.
- **SCAN**, la testina si porta ad una estremità del disco e si sposta verso l'altra servendo le richieste d'accesso man mano che vengono raggiunte le tracce indicate.
- **CSCAN**, è una variante di SCAN mirata a fornire un tempo di attesa medio dei processi più basso.